Chapter 9

Quality Assurance

"*Quality is not an act, it is a habit.*" — Aristotle

Software projects must operate under many different constraints. These constraints include such things budget constraints, resource constraints, product constraints, and organisational constraints. However, in many software projects, the processes and product must meet certain *quality standards*, which are constraints as well.

All the evidence that we have points to a situation where we cannot simply fix up our software *post-hoc* and add in quality attributes after building the system. Quality must be built into the software from the beginning. One can aim to improve the quality of a software product by spending a lot of time testing near the end of the project, but this will not be effective if the requirements were poorly elicited and specified. Nor will it be useful if the data structures and algorithms are incapable of achieving the performance constraints.

To achieve these quality standards in a software engineering project, we address these standards using *quality assurance*.

9.1 What is software quality?

Quality is a ubiquitous concern in software engineering. Achieving quality pervades software engineering processes, methods and tools, and is one of the chief aims of project management. Quality products can increase market share and long term profitability for companies, and poor quality can have the opposite effect. Yet, in spite of the focus on quality in software engineering, it can be remarkably elusive to attain in actual software engineering projects.

Ask ten different individuals what software quality means and you are likely to get ten different answers from ten different perspectives. However, we can categories the different perspectives based on the role that a person fulfils when interacting with a software product. There are three general role perspectives of software quality:

- **The End-user's Perspective** — Typically, end-users judge the quality of a product by their interaction with it. For users, a system has quality if it is fit for purpose, is reliable, has reasonable performance, is easy to learn and use, and helps the users in achieving their goals. Sometimes, if the functionality is hard to learn but is extremely important and worth the trouble of learning, then users will still judge the system to have high quality. These are termed *external quality characteristics*, because they are typically associated with the external behaviour of the system.
- **The Developer's Perspective** — The developer's perspective typically also includes the number of faults that the system has, ease of modifying the system, ease of testing the system, the ease of understanding the system design, the re-usability of components, conformance to requirements, resource usage, and performance. These are mainly *internal quality characteristics*, because they are concerned with the quality of the internal structure of the system.

Ultimately, it is the end-user's perspective that matters. To quote business writer and consultant Peter F. Drucker:

"*Quality in a product or service is not what the supplier puts in. It is what the customer gets out and is willing to pay for. A product is not quality because it is hard to make and costs a lot of money, as manufacturers typically believe. This is incompetence. Customers pay only for what is of use to them and gives them value. Nothing else constitutes quality.*"

From this, one can assume that the developer's perspective is irrelevant; however, this is not the case. The reason that we consider the developer's perspective is precisely because making our systems understandable by developers and maintainers increases the likelihood that our product will satisfy the end user. If a system is more straightforward to understand and maintain, it is less likely we will make mistakes, and we can focus out resources on increasing quality with respect to the end user.

One of the chief jobs of a software engineer is to choose processes, tools and techniques to monitor and control the quality of the software as it is being developed, and taking into account all of the different perspectives.

If a software engineer cannot directly monitor the quality attributes of the *product*, then they will often instead monitor the quality attributes of the *processes* being used to developed the product, under the assumption that *the quality of the process influences the quality of the product*.

Monitoring product quality typically involves *measuring* and *assessing* products and processes to be confident that they will meet stakeholders' expectations.

9.1.1 Quality models

Quality is difficult to measure or assess directly, because the quality of the product is dependent on the perspective, interpretation, and expectations of individuals. Ideally, we would like to remove such biases wherever possible and seek more objective measures of software quality. For this reason, software engineers use *quality models* that decompose the concept of quality into a number of attributes; each of which can be measured or evaluated more objectively. In fact, many of these are the same attributes that you would have seen earlier in this subject, and in subjects such as 433-294 and 433-320.

Let us begin with a relatively abstract characterisation of *quality*. For a system to have quality we usually require that it:

1. satisfies *explicit* functional and non-functional requirements — that is, it should be correct, complete and consistent with respect to the system's explicit requirements;

2. adhere to internal (organisational or project) and external standards imposed on the project — for example IEC61508 (safety), Rainbow Standards (US Security), IEEE standards, internal company standards; and

3. conform to *implicit* quality requirements, which are requirements for performance, reliability, usability, extendibility, safety, and security and typically capture what we think of as the attributes of quality.

Note: there are a number of reasons for choosing to *build* a set of specific attributes a system. The first is that there are specific non-functional requirements that we must adhere to because they are requirements[1]. The second reason for choosing to build a set of attributes into a system is that either our project has a set of standards that it must conform to, or that our organisation has a set of standards that we must meet.

If we accept this idea then we can start to analyse quality in terms of a set of standardised attributes. *Quality models* present a standardised set of measurable attributes that can be used to judge the quality of a system.

Our first example of a quality model, by McCall et al. [MRW77], is one of the first quality models developed, and is shown in Table 9.1. This model breaks the concept of a quality (software) system into the eleven *quality attributes* listed in Table 9.1. Note that the first five of these attributes are from the end-user's perspective, while the remaining six are from the developer's perspective[2].

Another example of a quality model is given in the ISO-9126 standard for software quality [ISO91], in which quality is broken down into the *quality attributes*, as shown in Table 9.2. ISO-9126 breaks the concept of quality down

[1]These requirements should take the form of quantifiable measurable targets such as a statement of *mean time to failure* for reliability, throughput, response time for performance, or *continuous delivery of service without failure* for availability.

[2]Portability may also be desirable for the end user.

Quality Attributes	Definition According to McCall et al.
Correctness	The extent to which a program satisfies its specifications and fulfils the user's mission objectives.
Reliability	The extent to which a program can be expected to perform its intended function with required precision.
Efficiency	The amount of computing resources and code required by a program to perform a given function.
Integrity	The extent to which access to software or data by unauthorised persons can be controlled.
Usability	The effort required to learn, operate, prepare input, and interpret output of a program.
Maintainability	The effort required to locate and fix an error in an operational program.
Testability	The effort required to test a program to ensure that it performs its intended function.
Flexibility	The effort required to modify an operational program.
Portability	The effort required to transfer a program from hardware and/or software environment to another.
Reusability	The extent to which a program (or parts thereof) can be reused in other applications.
Interoperability	The effort required to couple one system with another.

Table 9.1: McCall's quality model and its interpretation of external system attributes.

into attributes similar to those of McCall et al., but goes further by decomposing each attribute into *sub-attributes* that are intended to be easier to assess than the parent attribute. As with McCall's model, there are perspectives of both the end user (the first three attributes) and the developer (the remaining three attributes).

Assessing attributes in quality standards such as McCall's or ISO9126 is not so easy in practice, as some interpretation of whether or not an attribute meets its targets is still required. In some cases, such as usability, it is difficult to even specify the targets.

9.1.2 The software quality dilemma

The quote from Drucker earlier in this chapter provides a dilemma for software engineers. Drucker states that a quality product is what the customer gets out of it, not what the suppler puts in. Thus, a customer will pay only for the utility value they receive from a product, irrelevant of the time, effort, or cost that went into its production.

This chapter is all about software quality, and how to achieve it. It discusses some processes and techniques that can be used to assure quality in software. However, each of these processes and techniques comes at a cost, including the salaries of the people that perform them, the cost of software licenses and hardware of the tools used, and the extra time taken to apply them that may push back the release date.

However, the lack of quality in a product also has a cost. First, it costs the users of the system, who must deal with the low quality. This has a knock-on effect for the supplier, because potential users may choose not to purchase the software if the quality is low, and current users may not purchase upgrades. Second, there is a cost again to the supplier in maintaining the low quality software. Not only will fault reports be more common, but when changes are made to the system (correcting fault, adding new features, or some re-factoring), it is likely to take more effort and cost than for a high quality system.

The software quality dilemma asks: which process and activities are worthwhile to apply in order to achieve software quality?

The general answer is that, for every project, the software quality must be "good enough". The problem with this answer is that "good enough" is different for every project.

For example, in the case of a desktop calendar application for tracking meetings, the software may be considered

Characteristics	**Sub-characteristics**
Functionality	Suitability
	Accuracy
	Interoperability
	Security
Reliability	Maturity
	Fault Tolerance
	Recoverability
Usability	Understandability
	Learnability
	Operability
	Attractiveness
Efficiency	Time Behaviour
	Resource Utilisation
Maintainability	Analysability
	Changeability
	Stability
	Testability
Portability	Adaptability
	Installability
	Co-existence
	Replaceability

Table 9.2: The quality model introduced in ISO-9126.

good enough if the manage functions for managing a schedule and providing meeting reminders is reliable, but some of the lesser-used functions, such as exporting the calendar to other formats, is less reliable. On the other hand, a system for controlling the braking system of a train must be released with no known faults if the provider does not want to be negligent.

To answer the question for a specific project, management must have estimates for the cost of the quality assurance activities. They must also have estimates for the cost of the possible *risks* associated with different types of defects. Using these estimates, decisions must be made as to which quality activities will be pursued, and which will be foregone.

9.1.3 The cost of quality

Some software developers — and we explicitly distinguish between those who develop software, and the subset of these who *engineer* software — will claim that most quality assurance activities are too costly to administer. For example, instead of performing formal reviews of requirements specification documents, it is far better to build the system, ask the client/user for feedback, and to correct any faults from there. Alternatively, one can simply release the system and correct faults as users report them. The assertion is that the savings made by not using resources for quality assurance are greater than the costs incurred in fixing the faults.

However, studies indicate that this is not the case. A study published in 2001 from the Centre for Empirically-Based Software Engineering indicate that the later in the development lifecycle that a fault is detected, the more costly it is to locate and repair. Figure 9.1 shows the results of this study.

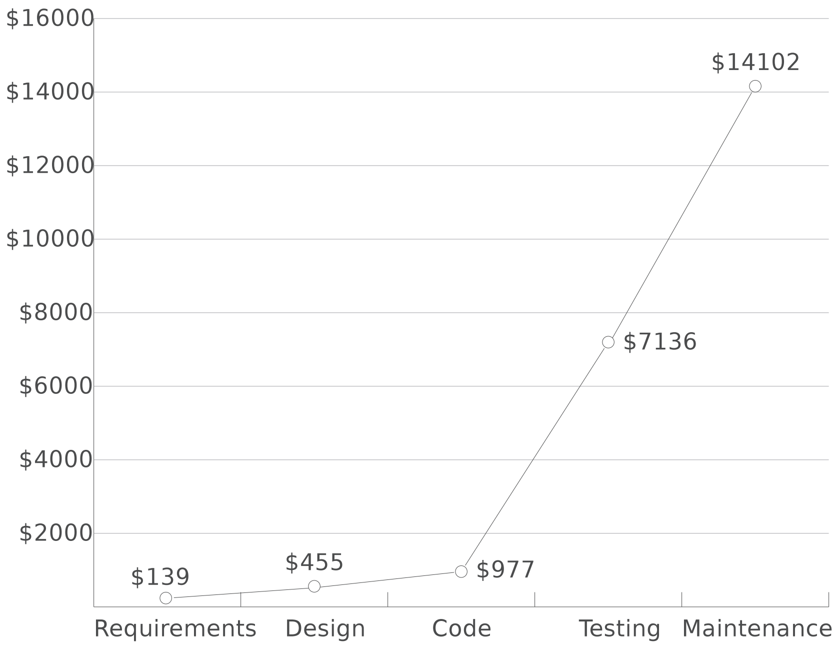

Figure 9.1: The cost a finding and fixing faults at different stages of the development lifecycle.

9.2 What is quality assurance?

The phrase “quality assurance” consists of two words: “quality” and “assurance”. The first word, “quality”, we are beginning to understand. But the second word, “assurance”, can be just as difficult. A general definition is as follows.

Definition 9.1. *Quality Assurance*
Quality assurance is the monitoring and evaluation of the various aspects of a project, service, or facility to ensure that standards of quality are being met.

Note that this definition includes nothing about guarantees, such as “my program is absolutely correct”. The aim of quality assurance to provide a high level of assurance, or a high degree of confidence, that your program will meet the needs for which it was written. The same is true for systems consisting of sets of programs and hardware.

Also note the difference between quality assurance and quality *control*. The latter is about testing a product before its release with the purpose of deciding whether to release the product. Quality assurance is the processes of improving the product quality during its development.

Remark: What is important in *engineering*, and one thing that distinguishes engineering from ad-hoc development, is your ability to exert control over the level of assurance achieved in projects. In practice, exerting this kind of control over quality means that we need to understand the quality attributes that we are interested in for our project, *and* that we have methods to build these attributes into our system.

9.3 Product vs. process

The two quality models presented in Section 9.1.1 both attempt to define software quality by listing the attributes that make up a quality product. However, a question remains: if we find that our projects are often lacking in one or more quality attributes, how do we change this? One answer is to get geniuses to build our products. However, it is not possible for all organisations to hire better developers — there are only a small handful of geniuses out there. A more reasonable answer is that we must change how we are building the product — in other words, we must change the *processes*.

This is where one of the key premises underlying process:

The quality of the product depends on the quality of the process.

Furthermore, this process can be defined, managed, measured, and improved. It is the hope that by doing so we improve the quality of our products that the process is used to produce. To improve the functional and non-functional aspects of a system, an innovative design or new technology may be used, but to improve the *quality* of a system, we need to improve its quality attributes — and this means employing a more suitable process.

The problem for us is that there is a complex and poorly understood relationship between software processes and product quality, so achieving quality is still not a straightforward exercise.

To achieve quality, we must be practical. A good tip is to decide which quality attributes are the most important for your project, and more effort and resources into developing those attributes — but do not neglect the others. To do this, a project team must:

- **Decide upon targets for process measures.** For example, acceptable defect rates from inspections of code or inspections of designs.
- **Decide upon targets for product measures.** For example the target reliability or performance estimates for you system.

If the targets that you set are not being met, you must do at least one of two things: 1) reduce your targets; or, preferably 2) change the processes that you are using so you can get closer to those targets.

Do not use use practices simply because the standard you have chosen requires it — understand why the practices are needed, and make sure that they fits with the goals of the project that you are working on.

9.4 Assuring quality with reviews and audits

In this section, we will looks at some of the key methods for assuring quality. The main methods for assuring quality are:

- **Technical reviews** — reviews of artifacts performed by peers in the development team with aim of uncovering problems in an artifact and seeking ways to improve the artifact.
- **Audits** — reviews of processes and teams to determine if a particular product or process conforms to standards.
- **Testing and Measurement** — dynamic execution of a system with the purpose of finding faults (testing functional requirements) and measuring whether the system meets its non-function requirements.

In this subject, we will discuss only the first two of these: reviews and audits. Testing and measurement methods are covered in detail in 4SWEN90006.

9.4.1 Technical Reviews

A technical review is a type of peer review that aims to find defects in a software artifact, or determine the quality of an artifact; for example, to decide whether the artifact is stable.

Due to them being "soft" methods for quality assurance — that is, nothing is executed — many developers greet reviews with skepticism, feeling that straightforward approaches can not be useful. However, empirical evidence suggests that such skepticism is unjustified for several reasons:

1. Reviews can be performed on *any* software artifacts, whereas many "hard" methods of quality assurance, such as testing and measurement, can be performed only on executable artifacts.
2. As summarised in Figure 9.1, earlier detection of problems in software artifacts leads to lower costs of resolution. Reviews can detect problems earlier than testing and measurement, due to point 1 above.

3. Due to internal pressure of getting software releases out the door, programmers make more mistakes when correcting faults that were found during testing than they do correct faults round during review, which indicates that, even for code, reviews can be useful.

4. The rate of fault detection using reviews is high. Even in source code reviews, roughly 30-70% of all programming faults found in a project were located using reviews, and up to 80% according to studies performed by IBM. Furthermore, one study [Mye78] demonstrated that review techniques found several *types* of faults that testing failed to find, and vice-versa.

5. Reviews find the actual faults in source code, in contrast to testing, which merely indicates that there is a fault somewhere in the program. After a fault is detected with testing, it must then be located.

An important part of a technical review is that it is a type of *peer review*. That is, the review is carried out by someone other than the author(s) of the artifact under review. Peer reviews are useful because the third party undertaking the review is likely to have different assumptions and a different understanding of the problem. For example, if the author of an artifact misunderstands a requirement, a self-review will not detect this. However, a peer review may.

You will probably have all experienced a case in which a third party has pointed out an obvious spelling or grammatical error in a document that you wrote, even though you checked it a number of times. Finding flaws in one's own work seems to be counter to the way the human brain works. As with writing, software artifacts will contain flaws, and the developers of those artifacts are less likely to find them than third-party reviewers.

Note: this is not to say that authors should never review their own artifacts. Quite the contrary. Authors most definitely should review their artifacts before passing them on for peer review.

9.4.2 Informal peer reviews

An informal review is a simple *desk check* or casual meeting with a colleague which aims to improve the quality of a document. There are no formal guidelines or procedures that are followed, and typically an informal review is simply asking a colleague to look over something and provide feedback. The effectiveness of informal reviews is considerably less than formal reviews, because of the lack of diversity found in a group, and because of the lack of focus by the reviewer.

Checklists are tools that can help to improve the effectiveness of a review. A checklist is a list of questions that the reviewer must answer about an artifact, however, the questions are generic questions about that *type* of artifact. That is, a generic checklist for requirements specifications is employed throughout an organisation. Checklists remind the reviewer of the key attributes that the artifact must fulfil, help to focus the reviewer on important aspects of the artifact, and provide a structured way to provide feedback. Checklists can be high-level or detailed, depending on the goals of the organisation. Figure 9.2 contains a lightweight checklist for a requirements specification artifact.

9.4.3 Formal reviews

A formal review is a meeting of a group of project stakeholders, such as developers, project managers, and clients, for the purpose of improving the quality of a software artifact, such as a requirement specification, or a piece of code.

A group meeting allows us to leverage from the diversity of the individuals within the group help improve the quality of an artifact. Furthermore, the group-based approach is more productive than having several individuals due to people's inherent want to feed their ego: in the meetings, people like to show off how smart they are by finding as many defects as possible.

Improving the quality of the artifact is achieved by: (a) uncovering logical errors within the artifact; (b) verifying that the artifact meets its specification; and (c) ensuring that the artifact achieves the requirements set out by some specified standards.

Review meetings

A review meeting should adhere to the following constraints:

Checklist for software requirements specification artifact

Organisation and Completeness

- ☐ Are all internal cross-references to other requirements correct?
- ☐ Are all requirements written at a consistent and appropriate level of detail?
- ☐ Do the requirements provide an adequate basis for design?
- ☐ Is the implementation priority of each requirement included?
- ☐ Are all external hardware, software, and communication interfaces defined?
- ☐ Have algorithms intrinsic to the functional requirements been defined?
- ☐ Does the specification include all of the known customer or system needs?
- ☐ Is the expected behaviour documented for all anticipated error conditions?

Correctness

- ☐ Do any requirements conflict with or duplicate other requirements?
- ☐ Is each requirement written in clear, concise, unambiguous language?
- ☐ Is each requirement verifiable by testing, demonstration, review, or analysis?
- ☐ Is each requirement in scope for the project?
- ☐ Is each requirement free from content and grammatical errors?
- ☐ Is any necessary information missing from a requirement? If so, is it identified as "to be decided"?
- ☐ Can all of the requirements be implemented within known constraints?
- ☐ Are any specified error messages unique and meaningful?

Quality Attributes

- ☐ Are all performance objectives properly specified?
- ☐ Are all security and safety considerations properly specified?
- ☐ Are other pertinent quality attribute goals explicitly documented and quantified, with the acceptable tradeoffs specified?

Traceability

- ☐ Is each requirement uniquely and correctly identified?
- ☐ Is each software functional requirement traceable to a higher-level requirement (e.g., system requirement, use case)?

Special Issues

- ☐ Are all requirements actually requirements, not design or implementation solutions?
- ☐ Are all time-critical functions identified, and timing criteria specified for them?
- ☐ Have internationalisation issues been adequately addressed?

Figure 9.2: Example checklist for a software requirements specification, courtesy of Karl Wiegers.

1. The review team should be between 3-5 people. Any more than this and there is too much communication and too many opinions to make the meeting productive.

2. The meeting itself should last no longer than 90 minutes. Humans typically do not have a long enough attention span to focus on a review task for longer than 90 minutes, so any longer than this, and the meeting becomes unproductive. This implies that teams should review only a part of the an artifact if it is too large to review in 90 minutes.

3. The following roles should be filled by a team member:

- **Review leader** — one person is responsible for organising the review, including finding reviewers, organising the meeting time/place, distributing checklists, and distributing the artifact to the reviewers, along with any supporting documentation; for example, a specification of the artifact.
- **Author** — at least one author of the artifact should be present.
- **Reviewers** — at least 2-3 reviewers who are not also authors.
- **Recorder** — one person is responsible for recording all important issues that arise during the meeting.

One person can fulfil more than one role, except for the person fulfilling the author role. Furthermore, if fulfilling two roles, one should be the role of reviewer — that is, one person should not be recorder and review leader.

The review starts with an introduction of the artifact from the author. Then, the author walks through the artifact in a logical manner, and explains the content in the artifact. For example, if the artifact was a piece of code, the author would step through each statement, saying what the intended behaviour of that statement is. At any point, one of the reviewers is permitted to raise issues based on their understanding. The recorder takes note of any important issues that are identified by the team.

An interesting note is that, during a review, many of the issues are raised by the author themselves. The process of stepping through their artifact in a logical manner and explaining it, forces them to think about it in a way they had not before.

After the walk through is complete, the review team must make one of the following recommendations:

1. accept the artifact without further changes;
2. accept the artifact with minor changes; or
3. reject the artifact, requesting non-trivial changes. In this case, the artifact must be reviewed again once the changes are made.

After the meeting, the recorder produces a report on the finding of the review, which lists the issues that were identified by the team. The author(s) make the changes, and typically produce a *response* document, which outlines where and how the issues were changed. This facilitates any subsequent reviews. Both the report and the response are submitted as part of the project records.

Walkthrough vs inspections

Walkthroughs and inspectors are both types of technical review that are discussed in software engineering literature. Both are similar types of review, in that they rely on groups to improve the quality of an artifact, and consist of the group walking through an artifact to find problems. The differences between them depend on who you ask. Some software engineers will tell you that there is no difference. The IEEE standard 1028–1997 specifies them as being similar, but not identical.

In this subject, we do not distinguish between walkthroughs and inspection, and instead simply model them together. Some notable differences that are commonly put forward, among others, are:

1. **Moderator** — in the case of a walkthrough, the role of the walkthrough leader is fulfilled by the author (or one of the authors) of the artifact, whereas in a review, it is fulfilled by another reviewer.
2. **Preparation** — walkthroughs not not require any preparation on behalf of the reviewers, whereas for inspections, reviewers are expected to inspect the artifact and take notes into the meeting.
3. **Taking action** — in an inspection, defects and inconsistencies should be identified, but not corrected, whereas in a walkthrough, possible solutions may be discussed.

Review tips

There are several tips that can be followed to ensure a good review:

1. **Use constructive criticism** — remember that the author of the artifact is present, and may have put a lot of work into it, so tearing it apart is disheartening, and may result in them rejecting the issues that are raised. Constructive criticism is difficult, however, one common technique employed in reviews is to ask questions so that the author can find the issues themselves, rather than point them out directly.

2. **Stick to the agenda** — a review meeting should be kept as short as is reasonable, and must keep on schedule. If the team decides that issues should not be solved during the meeting, then they should identify an issue and move on. A review will only be successful if the reviews are focused.

3. **Minimise discussion** — once an issue is raised, it may be that some reviewers agree with it, while others believe it is not a problem. If this is the case, simply note the problem so the author can return to it.

4. **Allocate time for reviews** — a project manager should explicitly schedule time for reviews, responses, and follow-up reviews. A review should form part of a quality gate (discussed in Section 3.4) for a process.

Review metrics

As with other software engineering activities, organisations benefit from understanding the effectiveness of performing reviews. To do this, they need to record metrics about the review process, the effort expended in performing the reviews, the effort expended reworking the artifacts, and the amount of defects that are found.

For a given artifact, if we know the total number of errors found, *Err*, the total effort, $Effort$, including the effort involved in preparation, reviewing, and reworking, and the size of an artifact, *S*, for example, number of pages or number of models, then we can calculate the error density:

$$\text{Error density} = \frac{Err}{S}, \tag{9.1}$$

and the rate of error detection:

$$\text{Rate of error detection} = \frac{Err}{Effort}. \tag{9.2}$$

For example, given a collection of 18 class diagrams over 32 pages, and 22 errors, the error density is 1.2 errors per class diagram, or 0.68 errors per page.

Over a number of projects, an organisation can collected these metrics, and can start to use them for planning and prediction. If over a large collection of projects, the average error density is calculated, then given an artifact of a particular size, we can provide a rough estimate on the error density of that artifact. For example, if we have an error density of 1.2 errors per class diagram, and we have an artifact with 32 class diagrams, then we can *roughly* estimate the artifact will contain about 38-39 errors.

Metrics can also be used to measure the effectiveness of reviews. It is not possible to measure this directly after the review; instead, we must wait until at least after further quality metrics have been collected, such as testing data.

Wiegers [Wie01] presents data from several major software companies on the cost effectiveness of inspections. These companies reported up to a 10 to 1 return on investment for inspections compared to testing, and resulted in earlier project completion. This is largely due to the cost of repairing a problem located during testing, compared to the much lower cost of repairing it if found earlier. The studies showed that additional effort is required earlier in a lifecycle/increment of a project, but that this easily recaptured later in the project due to the lower cost of testing and debugging.

A further advantage of collecting metrics is that they offer way to assess the value of reviewing an artifact *before* it is actually reviewed. In projects with tight deadlines, reviewing all artifacts rigorously may not be an options. Instead, we can use *sample-driven reviews*.

Using the error density metric, teams review only a portion of each artifact, and calculate the error density of each artifact using the results of the review. Based on these results, teams can estimate which artifacts are more likely to be error-prone by the error density of the samples, and can focus their attention on these. For example, given a collection of 80 class diagrams for a particular sub-system, a team can review 20 of these, obtaining an error density of 1.2 errors per class diagram. If a collection from another sub-system is sampled in a similar way, but contains on 0.7 errors per class diagram, this is evidence that the first sub-system is more error-prone, perhaps because it is more complex, or perhaps because it was developed by a less experienced team. If an artifact is estimated to be more error-prone, the team should spend more time reviewing this.

For sample-driven reviewing to be successful, the sample of each artifact must be representative of the entire artifact, and must also be large enough to draw meaningful measures.

9.4.4 Software audits

A software audit is a type of review, except that it does not aim to find defects in logic or meaning of an artifact, but only assesses whether a given artifact complies with a specified standard or process.

An important difference of an audit from a peer review or technical review is that the authors of the artifact being audited are not involved in the audit process at all. Audits are typically performed by a team that is completely external to an organisation.

The roles involved in an audit are similar to that of a review, except there is no author.

Audits can be broken into two broad types:

1. *product audits*; and
2. *process audits*.

A product audit assesses whether a product complies with a standard. The input to a product audit includes the artifact to be audited, and a standard to which is must conform. It is the duty of the auditors to assess whether the artifact conforms to the standard. For example, the artifact may be a module of source code, while the standard is a coding standard for the programming language in which the module is implemented. It is the auditors responsibility to check that the model conforms to the standard.

A process audit assesses whether a team is following specified processes. The input to a process audit includes the processes that the team should be following, along with any evidence that can be used to assess this. For example, a team of auditors may be requested to assess whether a development team is following the organisation-wide process of checking code into a repository. The audit team can access to repository logs to ensure that each new module committed is accompanied by a test driver, each commit contains a meaningful log message, and that each member of the team has run all systems tests before committing new code.

In contrast to technical reviews, audits are typically more objective: either a team is following a process, or it is not. It may not be straightforward to find evidence to assess this, but lack of evidence is usually a sign that the process is not being followed.

9.5 Improving quality via process improvement

To improve the quality of the products we produced, we must change the way we produce them — that is, change the processes used to produce. However, haphazardly making changes to processes in the hope that quality improves is unlikely to reap any rewards. Instead, we must improve the quality of our processes.

The *Capability Maturity Model* (CMM) [Hum90] is a model that aims to provide an assessment of the *process maturity* applied by organisations, such as how well the processes are defined, how well they are understood, and how well they are applied.

The model provides a rating scale of 1 to 5, as outlined below.

Level 1: Initial — at this level, organisations are following few processes consistently, and any success is largely based on individual effort and experience.

Level 2: Repeatable — at this level, basic processes are in place and used on projects, but any such processes are employed based on experience with prior projects.

Level 3: Defined — at this level, a standard software process is defined and documented for the entire organisation, and all projects use a tailored version of this, which must obtain approval.

Level 4: Managed — at this level, organisations measure the quality of their processes and products using established metrics. These can be used to help prediction of product quality, and to measure variation between different processes.

Level 5: Optimised — at this level, the organisation pro-actively uses metrics as part of a quantitative feedback mechanism to asses and improve processes. Causes of process weakness are identified and processes modified to prevent them from occurring in future projects.

The Software Engineering Institute (SEI), who first proposed the CMM, offer assessments of organisations, and rate process maturity of that organisation.

More recently, the SEI has proposed a the *Capability Maturity Model Integration* (CMMI), which is a meta-model for process improvement. Rather than judging all processes in an organisation at a particular level, the CMMI breaks these into 16 different *process areas*, such as requirements management, configuration management, and measurement & analysis. Each processes area is given a rating. This reflects the fact that, depending on the types of applications that an organisation builds, it may want higher maturity in some process areas than others.

References

[Hum90] W.S. Humphrey. *Managing the software process*. Addison Wesley, 1990.

[ISO91] ISO. Information technology – software product evaluation – quality characteristics and guidelines for their use, international organization for standardization. International Standard ISO/IEC 9126, International Electrotechnical Commission, Geneva, 1991.

[MRW77] J.A. McCall, P.K. Richards, and G.F. Walters. Factors in software quality, three volumes. The National Technical Information Service, 1977.

[Mye78] G.J. Myers. A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM*, 21(9):760–768, 1978.

[Wie01] K. Wiegers. *Peer Reviews in Software: A Practical Guide*. Addison-Wesley, 2001.